Multi-Threading: Taking Advantage of Intel Architecture Multiprocessor Workstations

June 1998

Order Number: 283037-001

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DISCLAIMERS AND RESTRICTIONS

Performance report notice

THIS TEST REPORT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Information in this document is provided in connection with Intel products. No license, express or implied, by Intel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The Pentium II processors may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

The hardware manufacturer remains solely responsible for the design, sale and functionality of its product, including any liability arising from product infringement or product warranty.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference:

http://www.intel.com/procs/perf/limits.htm or call (U.S.) 1-800-628-8686 or 1-916-356-3104.

SPECint95 and SPECfp95 benchmark tests reflect the performance of the microprocessor, memory architecture and compiler of a computer system on compute-intensive, 32-bit applications. SPEC benchmark tests results for Intel microprocessors are determined using particular, well-configured systems. These results may or may not reflect the relative performance of Intel microprocessor in systems with different hardware or software designs or configurations (including compilers). Buyers should consult other sources of information, including system benchmarks, to evaluate the performance of systems they are considering purchasing. For more information about SPEC95, including a description of the systems used to obtain these test result, and other information about microprocessor and system performance and benchmarks, visit Intel's World Wide Web site at http://www.intel.com or call 1-800-628-8686.

ABSTRACT

Multi-threading is introduced as a way to take best advantage of the processor or processors in an Intel Architecture technical workstation. Multi-threading concepts and terminology are introduced, and are contrasted with "multi-tasking". Emphasis is placed on writing applications having single executables that run on *either* single or multi-processor systems. Examples and "rules of thumb" are given of existing application usage which lend themselves well to multi-threading, as well as situations where multi-threading should *not* be applied. The programming interfaces (calls) which an application can use are introduced.

INTRODUCTION

Today's operating systems strive – and compete – to make most efficient use of a computer's resources. While much has been done to efficiently share a machine's resources among several tasks (multi-tasking), this "large grained" resource sharing is the most operating systems are able to do without additional information from the applications themselves. Recent operating systems provide mechanisms allowing an application to control and share machine resources at a "finer grain" - threads. This paper discusses how use of threads on multi-processor Intel Architecture workstations can improve an applications' performance, responsiveness and throughput.

Prior to the introduction of the Pentium® II processor (and its architectural predecessor, the Pentium® Pro processor), technical workstations featuring more than one processor were relatively expensive. Usually, these multi-processor systems were reserved for applications specially developed for those machines. As multi-processor machines have become more available, OS support for threading has become standardized, making this technique available and approachable by all application writers.

While this paper follows existing technical workstation market trends and places slightly more emphasis on Microsoft* Windows NT*, we also discuss features available from modern UNIX* OSes.

Goals

Like most programming techniques, the primary goal of *multi-threading* is to allow a user to take best advantage of the resources of a machine and its attached network. This technique is inspired by observations that - for much of the time - the majority of the resources of today's machines are *idle*, and the speeds (throughput, transfer rate, etc.) of the various parts of the machine vary widely. Threads express the work done by individual portions of a task (process), and allow for finer grained scheduling. Thus, the goals of multi-threading are improvements in:

- Resource utilization: Effectively and consistently use all available processing power
- Throughput: Mask delays due to slow peripheral devices (or other data producers) by enabling other work to be done in the mean time
- Responsiveness: Maintain excellent GUI responsiveness while other computation continues
- Communication: reliable, responsive cross-process or cross-net communication

and to do this in a way that assures:

- Scalability: performance that improves in proportion to added compute power
- Compatibility: same binary for single and multi-processors
- Portability: source code for an application varies minimally from platform to platform

Background and Terminology

MULTI-THREADING VS. MULTI-TASKING

Loosely speaking, multi-tasking refers to running multiple, unrelated jobs on one system or display unit – for example, a simulation and an e-mail client. Multi-tasking is by definition, "large grained parallelism"; independent jobs are executing at the same time. By contrast, multi-threading usually refers to a single job that is managing multiple, usually shared resources (memory, processors, etc.) at a finer grain size than multi-tasking.

Process as an Address Space

Today's operating systems model a running application – a task or *process* – as:

- an address space or work area where data can be manipulated,
- the executable code running within it, and
- descriptors for OS and machine resources used by the process.

The three interact with the OS to do the work of the task - receiving input data, transforming it and delivering it to some consumer or storage area. The executable code may have to spend significant amounts of time waiting: for I/O to complete, for higher priority jobs to run, for user input, for communication media, etc. Unless a task is very carefully written - or the machine is constantly busy with work for other tasks - processor and other machine resources can become idle.

Thread as a "Program Counter" Within the Process

All of the executable code for a process is brought into its address space. Applications usually do their work in segments or stages - modifying data for that segment or stage, and recording or sharing progress by updating global data. If the segments or stages of work operate without (or minimally) disturbing other units of work, each of these smaller units of work can run in parallel as *threads*. Each unit of work has its own program counter tracking the instructions being executed, and its own call/return stack - in essence, its own state.

DEFINITIONS

Address Space – a region of (virtual) memory that is protected from other address spaces and process on a machine, in which a task's executable instructions and data are stored

Process – synonym for:
Job – synonym for:
Task – An executing application that consists of a private virtual address space, executable code, data, and other operating system resources, such as files, pipes, and synchronization objects that are visible to the process. This includes call/return stack(s), shared objects, I/O handles and environment variables, program counter(s), etc. The "classic" OS process has *one* thread of execution doing its work.

Multi-tasking – the ability of a machine and OS to run multiple, independent processes - conceptually at the same time - by sharing a machine's resources among those processes, usually by some time-slice strategy

Symmetric Multi-Processing (SMP) – a computer system constructed so that every processor has equivalent, full access to machine resources - memory, peripherals, graphics and other controllers. Additionally, any unshared resources (like L2 cache) have mechanisms to inform all processors of any need to synchronize content. As a result, any processor

can perform OS work equivalently well. (This is a slightly more general definition than a different "SMP" - Shared Memory Parallel Processing.)

Thread – one flow of control through a program and that flow's current state (represented by a current program counter, a call/return stack and, occasionally, some thread-private data). A process has one or more threads doing its work.

Multi-threading – having more than one flow of control within a process, allowing parts of the process to be independently performed

Asynchronous I/O – an "indirect" form of threading where an application makes calls to special I/O routines that initiate an operation and return immediately. The application may proceed doing calculations on other data, and is expected to make a later I/O call which checks whether the I/O operation has completed. This programming paradigm is gradually being displaced by use of threads (which are more general). Instead of using special asynchronous I/O calls, a thread is used to make "regular" I/O calls and synchronize upon completion the same way all other threads do.

Producer / Consumer – data processing is often performed as a sequence of *filters*, each performing a localized, well-defined operation. In such chains, each stage reads (consumes) some incoming data, processes it and emits or writes (produces) data for the next stage. 3D geometry is a classic example of this, with each stage doing operations like scale, rotate, transform, hidden line removal, texturing, etc.

Scalability – is a measure of the performance of doing work on multiple processors, relative to doing the same work on a single processor. For example, if a given application run takes two seconds on a single processor, and the same application takes one second on two processors, then the scalability for this task is 2. However, since there is work done by the OS and by the task to coordinate and synchronize work (i.e. overhead), the scaling factor never reaches *m* (the number of processors). This limitation is described more fully by Amdahl's Law, which relates processors, parallelism and synchronization. The quality of an implementation of threading is often measured by its scaling factor.

Concurrency – running activities in parallel

Synchronization – coordination of all the individual work by each of a set of threads into some merged result. One example is awaiting for one thread to finish filling a segment of a buffer before another begins using the newly-buffered data. Synchronization is also necessary when any thread has to alter data visible to another thread.

Semaphore – a synchronization mechanism - usable across processes - that maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource.

Mutex – a specially-handled binary variable that all threads agree to use to guard access to a shared structure, handle or other resource. This differs from a semaphore in that there is only one owner of the mutex at a time, but is like a semaphore in that it is usable across processes. A thread can only manipulate the resource if it holds the lock on the associated mutex; other threads await the release of the mutex variable. (In POSIX.1c UNIX, a mutex may optionally be intra-process -only.)

Critical Section – a special form of a mutex that offers very low overhead and is usable *only* among the threads of one process. (This is somewhat more direct to code in Windows NT than UNIX.)

Race Condition – a bug in use of threading where the code of one thread "A" relies on another thread "B" to complete some action, but where there is no synchronization between the two threads. The process operates correctly if thread B "wins the race" by completing its

action before thread A needs it, but the process produces incorrect or varying results if thread A wins the race.

SHARED RESOURCES

A key difference between multi-tasking and multi-threading is how application resources are shared. The following table contrasts different resource sharing scopes:

Programming and Hardware Features:		
Visible Throughout a Machine	**Visible Throughout a Process**	**Individually Owned by Each Thread**
▪ name space for file system, network machines, named pipes, etc. ▪ multi-process shared memory ▪ memory for, and names of, semaphores and mutexes ▪ pipes	▪ address space ▪ loaded executable code ▪ Win32* *resources* like string and icon tables, loaded fonts, etc. ▪ heap-allocated memory file handles or OS descriptors (including sockets and shared memory) and their state ▪ the ANSI C `errno` ▪ process arguments and environment list ▪ global and file-scope data ▪ memory for critical section variables	▪ program counter call/return stack ▪ thread-local storage (on some OSes) ▪ state/content of CPU registers ▪ scheduling priority ▪ UNIX signal masks ▪ various runtime statistics - CPU time, etc. ▪ ownership of mutexes and critical sections (when locked by that thread)

Table 1: Resource Sharing Scope

In most cases, this resource sharing can be of significant value; for example, all threads of a process can share work done to set up in-memory (heap allocated) data structures. With traditional multi-tasking, additional work – inter-process communication, shared memory, semaphores, etc. – must be done to arrange such sharing.

Care must be exercised, however: applications can be written to assume *no* sharing is happening. Updating such applications to use threading can result in unsynchronized changes to such shared data and the threads seeing inconsistent data. This risk can be minimized by:

- re-coding global data as a set of structures, each gathering related data
- guarding access to these structures by mutexes
- use of *thread-safe* libraries of functions and macros

ENABLERS

Multi-threading requires that the underlying machine and OS supply certain features allowing applications to coordinate or describe their activities. These include:

- primitives on which to build safe synchronization methods
- protection of tasks from one another – controlling object sharing
- hardware allowing multiple processors to communicate and identify themselves and their saved state
- primitives describing start, termination, synchronization and control of individual units of work

Processor Hardware Readily Available

These enablers are now available in workstations based on Intel architectures:

PROCESSORS AND CHIPSETS

- a wide variety of interlocked compare-and-exchange instructions
- built-in cross-processor communication, allowing "*glueless*" multi-processing
- cache consistency can be guaranteed in multi-processor systems, and synchronization variables can be safely held in cache, via "snoop" hardware assuring cache coherency
- processor state is easily saved and restored, including a processor identifier
- virtual memory hardware protects processes from each other
- high speed (100 MHz), wide (8 byte) system busses implemented by advanced chipsets supporting multiple processors

The incremental cost of getting a second processor is a fraction of the whole-system cost - often around $ 1,000.00 for today's workstations.

If a 2nd processor allows tools to run an average of 30% faster, that amounts to about 1 man-day per week of improved user productivity.

SYSTEM MOTHERBOARDS

Multi-processor motherboards are widely available – frequently the standard offering for technical workstations. Cooling, mechanical and electrical requirements for these boards are very similar to existing system-integrator experience, facilitating ease of use.

COMPLETE SYSTEMS

In the technical workstation marketplace, multi-processor systems are becoming the norm. Most suppliers of Intel-based systems routinely offer such machines to their technical customers.

Less Expensive memory

Multi-threaded applications place more demand on the memory subsystem. They require:

- **More memory** to save the state of the various threads - primarily its call/return stack.. With complex workstation applications, this state can become sizable – many megabytes. Additionally, more heap allocations are simultaneously present.
- **More memory bandwidth** due to a higher number of active memory transactions per second

Today's Intel hardware supports both at a low cost.

Multi-threading –capable OSes and System Software

Operating systems now supply well-defined methods for applications to start, control, terminate and synchronize *threads* of execution within one task. These individual units of control can share resources (like memory and I/O devices) and can communicate easily and cheaply. There are two major families of thread implementations – that of Windows NT and UNIX; they are *not* equivalent, but offer very similar services. Both are available on Intel-based systems. (A table on page 17 lists and compares the threading calls of the two OS families.)

WINDOWS NT

The Win32 Application Programming Interface (API) provides a complete set of threading primitives which is identical across all platforms implementing Win32.

THREADS IN UNIX

Threading on UNIX systems is a bit more complicated: there are two primary threading APIs available today:

- the POSIX*.1c threading standard, agreed-upon by all major UNIX implementers, and a feature of UNIX/98, now being standardized by the members of *The Open Group*
- "UNIX International" or "Solaris™" threads, available on some platforms, is a *superset* of POSIX.1c threads

An application written to POSIX.1c threads will be portable across UNIX variants.

PORTABILITY LIBRARIES

Besides each OS' native interfaces, API libraries are available from third parties that present portable interfaces to the programmer. While applying them is not automated, code written to those APIs becomes portable across Windows NT and most UNIX systems. Some examples:

- Threads.h++ from Rogue Wave Software http://www.roguewave.com
- Mthread from the book *Multithreading Programming Techniques* by Shashi Prasad; download MThread from http://165.254.151.1/people/shashi/book/mpt.html

As a special case, implementations of the Win32 API are available on UNIX, and of POSIX.1c threads on Windows NT.

APPLYING MULTI-THREADING

Many applications are currently written single-threaded. They can offer the user improved performance and responsiveness were they to be threaded. Key examples in this category include applications that spend most of their time doing array mathematics and region-by-region processing of graphical images.

Because of Amdahl's law - bounding the parallelization-based improvement of an application by the portion that *cannot* be parallelized - some applications are not good candidates for threading. A classic example is any application whose data merge or synchronization stages are larger than all the potentially parallel work to be done.

Here are some guidelines, some illustrated examples where threading is a "win" and statistics from commercially-available threaded applications.

When to Use Multi-threading

- Algorithms that can be independently applied to segments within a large data set (rows or columns of arrays, bands or regions of 2-dimensional data, etc)
- Pipelines with stages having similar amounts of work
- Maintaining GUI responsiveness even while long computations are ongoing
- When I/O can occur in parallel with processing of earlier-buffered data
- Multi-thread the parts of your code that consume the most time; they *often* are the easiest to parallelize and scale best

When not to Use Multi-threading

Although multi-threading is usually beneficial, there are some situations where it is not advised:

- Parallel tasks which would each do only a small amount of work (compared to the time needed to synchronize or merge the parallel tasks' operations)
- Tasks where synchronization overhead is as large, or larger than the parallelized execution
- Algorithms that change significant amounts of global state with each iteration (You can try to fix this in your application, then look at multi-threading it.)
- Applications having only isolated regions that could be parallelized

Additionally, if a workstation solution is assembled from several independent applications, multi-threading is not applicable. A cooperative multi-tasking strategy might be appropriate in this case.

Multi-threading Speedup Statistics from Real Applications

Vendor	Application	Job/Workload	Scaling Factor	
			2 processors	4 processors
SOFTIMAGE®	mental ray renderer	"harley"	1.92	
	Digital Studio	D1 video stream edit	1.54	
Intel Corp.	DGEMM with Intel's Math Kernel Library (MKL)	1000x1000 matrix	1.97	
NAG	LAPACK	2000x2000 matrix	1.84	3.12
	LINPACK	1000x1000 matrix	1.58	
3D modeling application (release from ISV is pending)		shading of a 3D object with a complex surface	1.9	
Fluent	(computational fluid dynamics)	"ugm2"	1.9	
MacNeal-Schwendler Corporation (MSC)	NASTRAN* v70	Bcell14a	1.33	
		lgast	1.57	2.02

Table 2: Multithreading Scaling with 2 and 4 Processors

See also a companion Intel White Paper *"Case Studies in Moving Multi-Threaded Workstation Applications from RISC/UNIX to the Intel Architecture"*, published June 1998.

Examples of Some Techniques

MASKING DELAYS OF SLOW PERIPHERAL DEVICES

Although today's hard disks are quite fast, processor speeds have advanced more rapidly. As a result, there are (roughly) 100's of "spare" cycles of processor time for *each byte* read off of

disk. The processor usually spends this spare time in an "idle" loop in the OS or in some other, lower priority application - instead of doing productive work for the application.

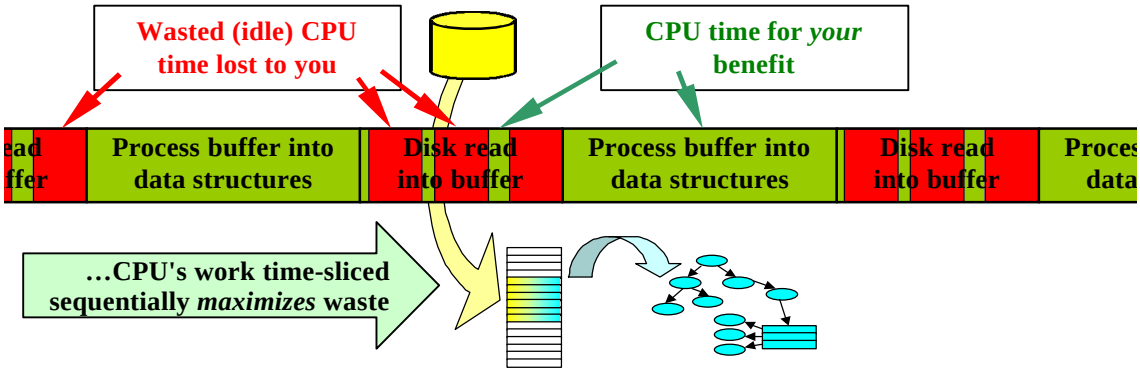

Figure 1: Processor cycles wasted by single-threading

You can overlap CPU processing with the otherwise-wasted I/O wait time by using threads or "asynchronous" I/O. Overlapping processing and I/O usually involves double buffering; at the modest cost of today's inexpensive memory, you can cause the processor to spend *most* of its time doing work for your application.

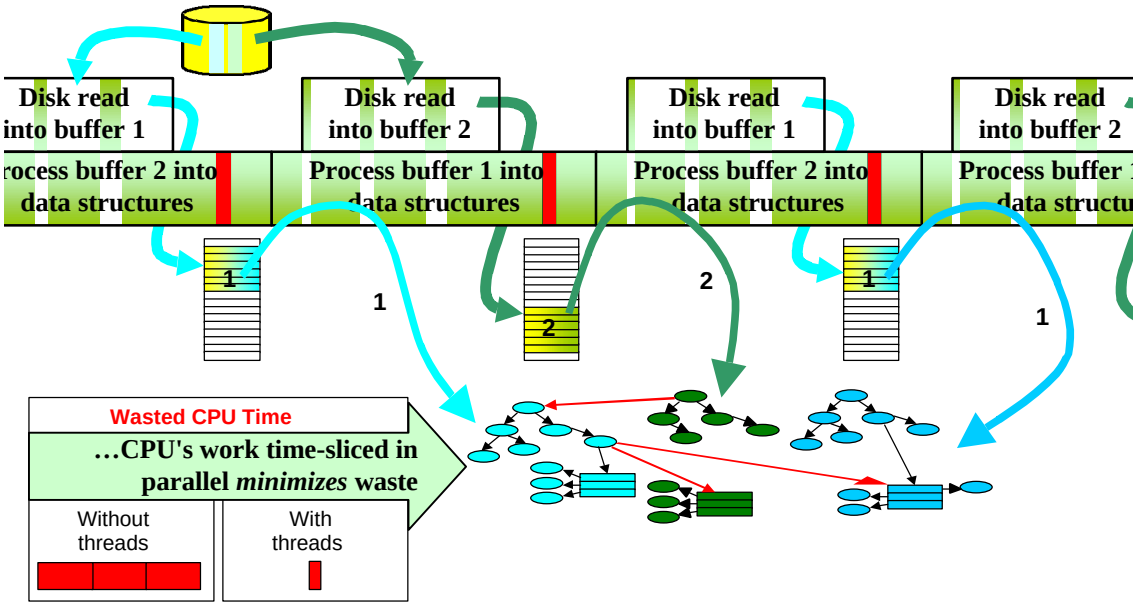

Figure 2: Processor cycles gained back by multi-threading

The example above shows how you can regain otherwise idle cycles of a processor. Double buffering can be easily extended to additional buffers. However, the time to complete the buffer read and the processing of the buffer must remain balanced; if not, a full buffer can waste time awaiting processing resource. Adding an additional processor can provide that compute

resource, gaining additional application performance improvement *with no further source code change*.

This example also illustrates that employing multi-threading can be of significant aid *even on uni-processor system*s, because otherwise-idle CPU cycles are used to do work while the I/O subsystem proceeds independently.

MAINTAIN GUI RESPONSIVENESS DURING COMPUTATION

Even when applications perform long computation sequences, users still want the application GUI to be responsive – if for no other reason than to control or stop an incorrect operation. In the absence of threading, the application has to resort to unpredictable mechanisms like timers or asynchronous `signal()`s, sporadic pauses in computation to check for input events, etc. While these mechanisms can be made to work, they usually result in:

- undesirably complex checks for external events injected into otherwise clean algorithmic code
- irregular or unpredictable GUI responsiveness

Largely for these reasons, today's GUIs are multi-threaded. For example, applications built with the Microsoft* Foundation Classes (MFC) are actually multi-threaded – though the application coder is often unaware of this. On multi-processor machines, GUI behaviour is almost flawlessly smooth.

EFFECTIVELY USE AVAILABLE PROCESSING POWER

Many algorithms are applied serially on large data sets in memory. If applying the algorithm to the data results in (predominantly) localized changes in the data, the application can be parallelized.

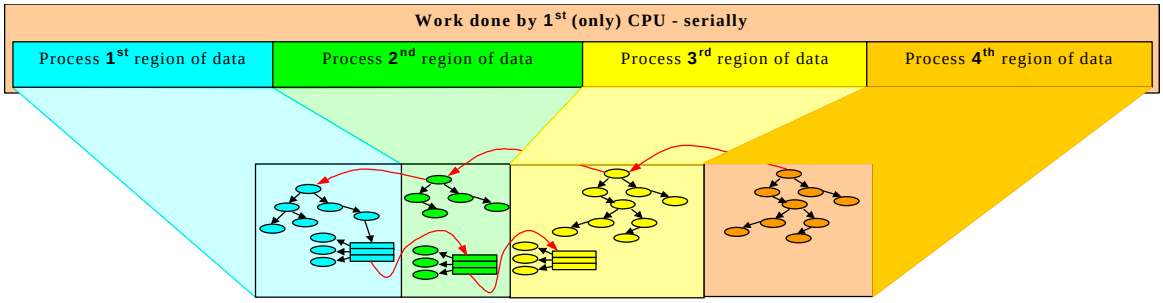

Figure 3: Serial Data Processing

Multi-threading allows a second processor to operate on the data structures in parallel with the first processor. As with all parallelized operations, there is some overhead due to:

- thread start-up and state retention
- checks in the algorithm that operations are staying in the localized data, postponing or queuing cross-thread operations
- intermediate synchronization, if any
- final synchronization and merging of results

The example below illustrates several of these concerns, but still produces an overall improvement in processing time of some 45% - significantly more if additional processors were available.

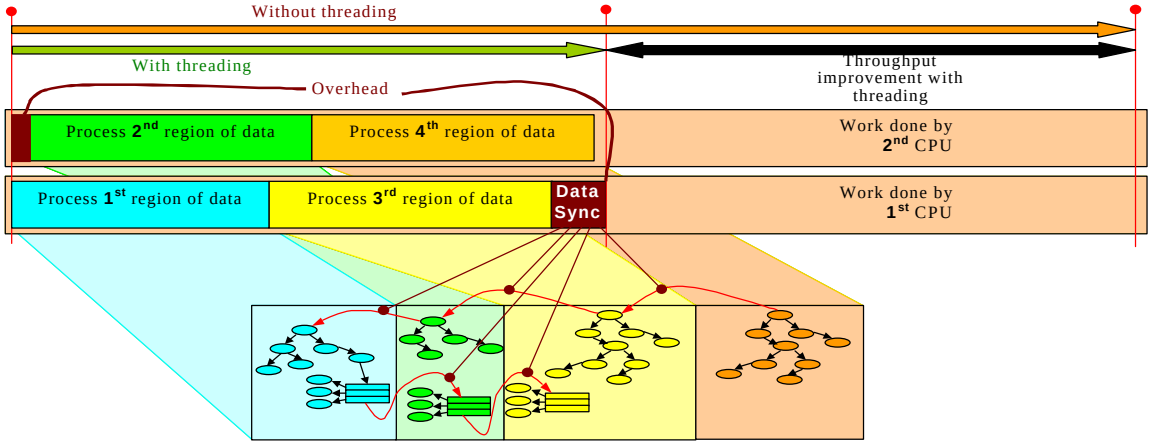

Figure 4: Parallelized Data Processing

This technique can be applied to many workstation applications and drivers:

- Fault simulation (which is highly localized)
- Stress analysis (after meshing has happened)
- Many graphical algorithms where data is coordinate sorted (processing can be done in *bands* or regions); for example IC CAD design rule check
- Many types of array calculations

IMPROVING PRODUCER/CONSUMER SEQUENCES

Some applications consist of sequences of data transformations. Without threading, the applications run each stage in sequence, completely buffering each intermediate data transformation:

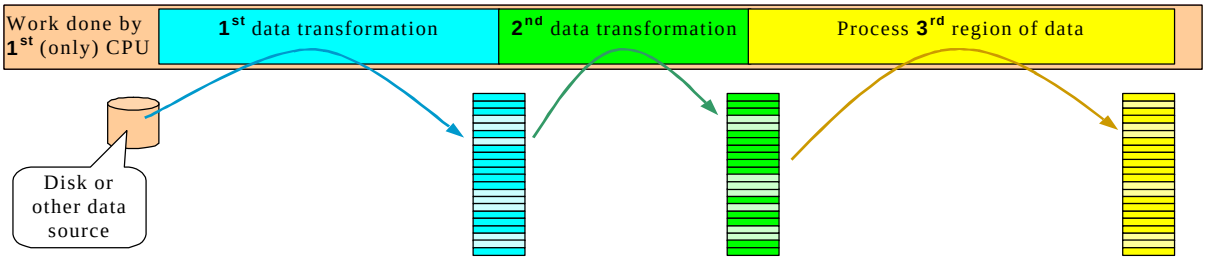

Figure 5: Single-threaded Producer/Consumer

By employing multi-threading, the overall throughput can be improved (here, by roughly 40%), and the intermediate memory consumption can often be reduced (with corresponding cache and swapping benefits):

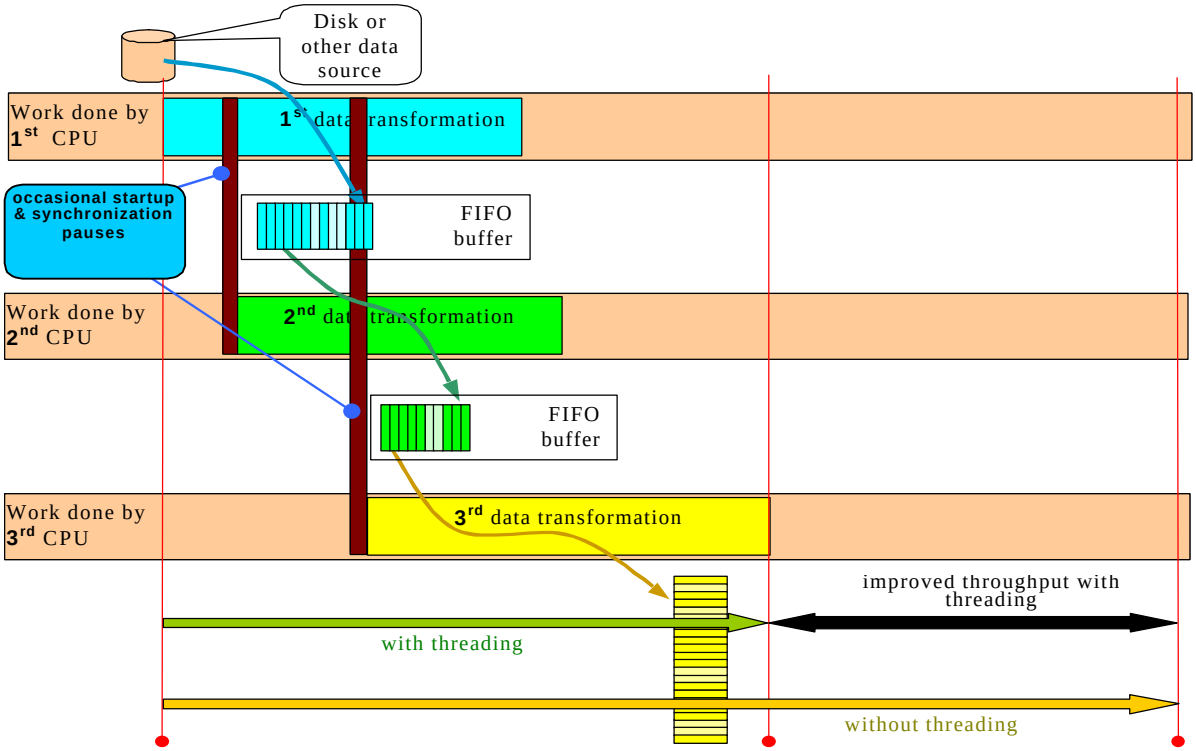

Figure 6: Multi-threaded Producer/Consumer

The cost of added synchronization and buffer management is usually significantly lower than the advantage gained in parallelization. These kinds of problems easily attain more improvement with additional processors, up to the depth of the pipeline.

Resources to Build Threaded Applications

To build a threaded application, the OS must supply mechanisms to:

- query machine capabilities (number of processors, etc.)
- start, exit from and terminate threads
- control thread scheduling and priority
- synchronize access to shared resources
- handle asynchronous events and signals
- accurately time (or time*out*) operations

These facilities can be used directly - or indirectly - from your application source code

AUTOMATED PARALLELIZATION TOOLS

Automated parallelization tools are becoming more common. These tools examine application source for actions that can be independently performed and loop iterations (particularly nested loops) that can be done in parallel. They transform your source by reordering operations and inserting compiler directives or API calls to specify start, synchronization, join and termination of parallel tasks. While these tools can do some parallelization autonomously, the application benefits greatly if the developer provides "hints" (directives) in the source code. For example, see:

- Applied Parallel Research; http://www.apri.com
- Kuck & Associates; http://www.kai.com
- Pacific Sierra Research; http://www.psrv.com
- The Portland Group; http://www.pgroup.com

To date, most of these tools work on FORTRAN source code, but standardization efforts are underway to make parallelization directives portable and usable from within `C` and `C++` also. Visit the *OpenMP* web site for more information, including specifications:

- OpenMP Consortium; http://www.openmp.org

Independent Software Vendors (ISVs) can benefit from using directives in 3 ways:

1. directives are a form of higher-level programming which can speed application development. Programmers still need to know their code, but parallel decomposition can be done more quickly than "hand coding".
2. directives like OpenMP are *portable*; coding done once will work on multiple platforms.
3. directives allow the compiler to do the *bookkeeping* work of threading and synchronization – one of the most error prone coding tasks.

THREADING API FEATURES

While automated threading tools can produce acceptable results, they work best only for "classic" multi-threading styles – nested loops, independent basic blocks, etc. Applications have greatest flexibility and control if they make threading calls directly. The table below describes many of the threading-related calls available to the users of Windows NT and POSIX.1c-compatible UNIX: (See the Win32 API documentation and *The Single UNIX Specification, Ver. 2* for more complete information.)

OS Family: / Purpose:	Windows NT	UNIX (POSIX.1c)
create thread	`CreateThread()`, `_beginthread()`	`pthread_create()`
thread terminates itself	`ExitThread()`, `_endthread()`	`pthread_exit()`
suspend and resume a thread to do work	`SignalObjectAndWait()`, `SuspendThread()`, `ResumeThread()`	`pause()`, `sigsuspend()`, `pthread_kill(…,SIGSTOP)`, `pthread_kill(…,SIGCONT)`
thread attribute control	`GetThreadPriority()`, `SetThreadPriority()`, `SetThreadPriorityBoost()`	various `pthread_attr_…()` calls

OS Family: / Purpose:	Windows NT	UNIX (POSIX.1c)
await thread termination	the various general-purpose …Wait…() functions - like SignalObjectAndWait(), WaitForSingleObject(), WaitForMultipleObjects()	pthread_join()
kill a thread	TerminateThread()	pthread_cancel()
thread explicit yield	SwitchToThread(), Sleep(0)	pause()
semaphore manipulation	CreateSemaphore(), OpenSemaphore(), the various general-purpose …Wait…() functions, ReleaseSemaphore()	sem_init(), sem_open(), sem_wait(), sem_trywait(), sem_post(), sem_destroy(), sem_unlink()
mutex manipulation	CreateMutex(), OpenMutex(), the various general-purpose …Wait…() functions, ReleaseMutex(), CloseHandle()	pthread_mutex_init(), pthread_mutex_destroy(), pthread_mutex_lock(), pthread_mutex_trylock(), pthread_mutex_unlock()
critical sections	InitializeCriticalSection(), Initialize…AndSpinCount(), DeleteCriticalSection()	*use in-process options on mutexes*
create and manage thread-local storage	TlsAlloc(), TlsSetValue(), TlsGetValue(), TlsFree()	pthread_key_create(), pthread_setspecific(), pthread_getspecific(), pthread_key_delete()

Table 3: Principal Threading-related Calls

Warnings, Risks and Remedies

At first glance, the risks and problems of multi-threaded code appear formidable. However, this technology has been available for many years and the risk and remedies are well-known. An observant programming team can easily review code to minimize these risks.

rapid creation and destruction of threads – Because thread creation and termination can be expensive, use a pool of pre-created threads that "sleep" until they are told what to do. This can be especially valuable in Windows NT "services" (in UNIX, "daemons") that use threads to connect to a requestor, hold the state of the request, and send back some result. The incoming request can be quickly delegated to a pre-created thread.

many more ready-to-run threads than processors available – the overhead of scheduling the threads becomes significant. (By contrast, the number of threads sleeping on programmed-for events is less of an issue.)

producer/consumer pipelines with unbalanced work – if each stage in a producer/consumer pipeline is *not* doing similar amounts of work, the pipeline's overall performance will settle to the throughput of the slowest stage

link applications with thread-safe libraries – many bugs are due to linking a multi-threaded application with the "standard" (i.e. *not* thread-safe) versions of the C library, etc.

***still* holding a mutex when a thread waits for something else** – holding a mutex blocks any other user of the object the mutex is guarding. If you are still holding a mutex when you

are blocked by something else - a slow I/O operation, a network communication, etc. - performance is lost. This also places you at risk of *deadlock* (see below). Do mutex locking at as fine-enough grain to avoid having to hold a lock while sleeping for something else.

race conditions – if a thread writes result data that affects the computations of another thread, access to the first thread's result data must be guarded by some synchronization mechanism. If it is not, an application will produce one result if the first thread "happens" to get done before the 2nd reads the data, and a different result if the first thread happened to take longer on a different run. It is important to develop QA test cases for a variety of load extremes.

deadlocks – if a thread holds a synchronization lock "A" when it "goes to sleep" awaiting on another synchronization object "B", there is some risk that another thread holding the lock "B" will go to sleep awaiting the first lock "A"; this is a classic deadlock. A common remedy for such deadlocks is to make the locks guard "smaller" objects, holding the locks for smaller amounts of time.

`signal()` or event mis-delivery – UNIX signals and many GUI events are delivered to the *process*; if a particular thread has not been coded to receive or handle these events, they can be lost or mis-delivered to whatever thread happens to be running at the time. This can be corrected by having a designated thread receive these asynchronous events; set thread-specific signal delivery masks or event handlers.

failure to check API return values – most threading APIs check parameters and process state, and return error indications if they detect any problems. Because multi-threaded applications are more complicated to test and debug, you can help yourself dramatically by coding checks for these API-reported error conditions.

shared data must always be guarded – an audit of an application for global or file-scope data that might be shared is very important. Many errors are made in assuming that such data is not shared, and "saving time" by omitting a synchronization guard that later turns out to be important.

Other problems, and techniques to avoid or correct them, are described in the references below.

REFERENCES

On-Line References

- *An Introduction to Programming with Threads*, by Andrew Birrell; http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-035.html
- News:comp.programming.threads, FAQ @ http://www.serpentine.com/~bos/threads-faq
- Win32 API documentation is available on-line to MSDN subscribers at http://premium.microsoft.com/msdn/library
- *The Single UNIX® Specification, Version 2*, by The Open Group http://www.rdg.opengroup.org/onlinepubs/7908799/toc.htm
- Applied Parallel Research http://www.apri.com
- Kuck & Associates http://www.kai.com
- Pacific Sierra Research http://www.psrv.com
- The Portland Group http://www.pgroup.com
- OpenMP Consortium http://www.openmp.org
- Rogue Wave Software http://www.roguewave.com

Books

- *Programming with Threads*, by Kleiman, Shah and Smaalders; ISBN 0-13-172389-8
- *Threads Primer: A Guide to Multi-Threaded Programming*, by Lewis and Berg, ISBN 0-13-443698-9
- *Multithreading Applications in Win32*, by Beveridge and Wiener; ISBN 0-201-44234-5, see http://www.awl.com/cseng/titles/0-201-44234-5
- *Win32 Multithreaded Programming*, by Cohen and Woodring; ISBN 1-56592-296-4, see http://www.oreilly.com/catalog/multithread
- *Multithreading Programming Techniques* by Shashi Prasad; ISBN 0-07-912250-7, see http://mcgraw-hill.inforonics.com/cgi/getarec?mgh27812